ToolCoder: Enabling Code Generation Models to Use Unknown APIs with API Search Tools

Anonymous ACL submission

Abstract

Automatically generating source code from natural language descriptions has been a growing field of research in recent years. Invoking correct APIs is crucial to code generation. However, existing code generation models struggle in handling unknown APIs (e.g., user-private projects and libraries) , often generating erroneous or even non-existent APIs. Inspired by the process of human developers using code search tools to learn unknown APIs, we propose ToolCoder, a novel approach that integrates API search tools with existing models to assist code generation and API selection. ToolCoder automatically invokes API search tools to retrieve relevant APIs and learns API usages from retrieved results. Our experimental results demonstrate that ToolCoder exhibits excellent performance and generalization ability across five public and private library code generation benchmarks, with at least 6.21% improvement on average pass@1 metrics and 9.64% improvement on average pass@10 metrics compared to state-of-the-art methods. Furthermore, we show that our relatively small ToolCoder model is comparable to one of the current best models, *i.e.*,, GPT-3.5, highlighting the potential of incorporating programming tools into the code generation process.

1 Introduction

Deep learning techniques have shown great promise in generating high-quality source code from natural language requirements. Nowadays, pre-trained language models have achieved the state-of-the-art results on multiple code generation benchmark, such as CodeX (Chen et al., 2021), ChatGPT (Chen et al., 2023; Ouyang et al., 2022) and CodeGen (Nijkamp et al., 2022) models.

Application Programming Interfaces (APIs) are crucial components of programs, and selecting correct APIs is a key step to code generation. Existing models demonstrate impressive proficiency in generating code and APIs through their extensive general knowledge. However, these solutions encounter significant challenges when faced with unknown APIs, especially those from private projects and libraries. The lack of specialized domain knowledge in dealing with such APIs often leads to the generation of erroneous code. These models may fabricate non-existent APIs, a phenomenon commonly referred to as "hallucination" of large language models (Ji et al., 2023).

We conduct a preliminary experiment to validate the "hallucination" for APIs. We apply a popular code generation model - CodeGen-2B to generate programs involving private libraries (Zan et al., 2022a). These private libraries contain numerous unknown APIs that code generation models have not seen before. We found that more than 90% of generated programs contain incorrect APIs. The above limitation hinders the application of code generation models in real-world software projects.

To assist models in generating unknown APIs, we draw inspiration from human programmers' code search behaviors. When encountering unfamiliar APIs, programmers turn to search engines to retrieve APIs relevant to current requirements or learn APIs through documentations. Inspired by this observation, our motivation is to enable code generation models to use search tools to obtain suitable APIs in those unknown libraries.

In this paper, we propose ToolCoder, a low-cost and efficient solution that integrates API search tools into pre-trained code generation models, mimicking the searching behaviors as in the demonstrative example. To enable models to use tools, we fine-tune these models upon the source code data containing tool usage information. To obtain the fine-tuning data, we propose an automated data annotation method, which uses ChatGPT to annotate tool usage information in the original source code. After the fine-tuning, we successfully integrate API search tools into code generation, allowing existing models to use external tools autonomously.

Private Library		Public Library		
Monkey	BeatNum	TorchData	Pandas	Numpy
96.72%	90.18%	62.70%	32.30%	26.34%

Table 1: Hallucination rate of CodeGen-2B model upon different libraries.

We carry out in-depth evaluations to show that ToolCoder is capable of improving pass rate (Chen et al., 2021) when dealing with unknown libraries and APIs. ❶ We evaluate our model on two private library benchmarks (Zan et al., 2022a). With the help of tools, ToolCoder can generate more unseen APIs and alleviate the hallucination in APIs, even outperforming GPT-3.5. ❷ We further evaluate ToolCoder on three public library benchmarks (Zan et al., 2022a). Our model achieves significant improvements over state-of-the-art API-oriented baselines (Zan et al., 2022b,a), with at least 1.39%, 3.26% and 10.11% pass@1 on the three benchmarks correspondingly. ❸ We also conduct an ablation study to analyze the different settings in our experiments, including the dataset, the training process, and the inference settings. To the best of our knowledge, this paper takes the first step to explore the concept of incorporating programming tools into code generation models.

Our contributions in this paper can be summarized as follows:

- We propose to enable code generation models to invoke code search tools, *i.e.,* ToolCoder, addressing code generation with unseen APIs.
- We propose an automated data annotation method in software engineering community and release a tool-augmented code generation dataset for further study.
- We conduct extensive experiments on code generation tasks involving multiple private and public libraries. We demonstrate that ToolCoder significantly reduces the frequency of generating erroneous fabricated APIs.

2 Motivating Examples

In order to demonstrate our motivation, we present the limitations of existing models in generating unseen APIs and how API search tools alleviate the limitations.

2.1 Hallucinations with unknown APIs

APIs are essential to modern software development. However, generating the proper API remains challenging for code generation models. Existing models often encounter the issue of "hallucination" (Ji et al., 2023) when faced with unknown or unfamiliar third-party library APIs.

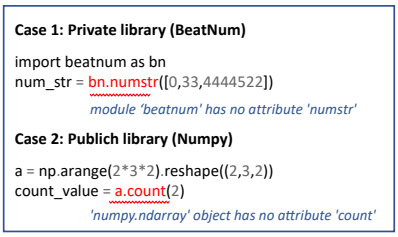

Figure 1: Hallucination examples of *CodeGen-2B* model in generating APIs, including unknown private libraries (*Case 1*) or even popular public libraries (*Case 2*).

To demonstrate this issue, we employ the popular CodeGen-2B (Nijkamp et al., 2022) to evaluate its API generation performance. We employ benchmarks from (Zan et al., 2022a) and generate 10 samples for each question. ***"Hallucination Rate"*** presents the percentage of incorrect API generations that are identified as hallucinations and verified to be incorrect in the respective third-party library. Table 1 reveals that over 90% of the generated APIs are incorrect in private (or unknown) private libraries, while even when dealing with popular public libraries, the incorrect rate still exceeds 26%. Figure 1 presents two examples of hallucination in the generated code for such APIs. Case ❶: CodeGen-2B lacks corresponding private knowledge and generates an incomprehensible API for the unknown private library *BeatNum*. This highlights the potential risk that existing code generation models may fabricate non-existent APIs for unknown APIs. Case ❷: Even for widely-used third-party libraries, the models may still struggle in API generation. *E.g.,*, CodeGen-2B generates a non-existent *count* API for the *NumPy* library.

The performance gap caused by APIs, particularly unknown APIs, is huge and quite severe, according to Table 1. It is essential to address these API generation challenges and propose solutions to finally improve generation quality.

2.2 Existing search tools to aid API generation

When facing unknown or unfamiliar libraries, it is natural for programmers to seek help and advice from API search tools, which inspires the design of ToolCoder. For example, the developer turns to **online search engine tools** or **library documentation search tools** and gets the proper API

	Online Search Engine	Documentation Search
Knowledge Resources	Programming Community or Tutorial Websites (StackOverFlow, datagy.io, etc.)	Library Documentation
API Type	Public libraries, especially those well-known and widely-discussed	Any APIs, including public and private libraries
Advantages	Practical and Accurate Rich sources Keep updating	Wide coverage Detailed explanation Stable
Example Tools	Google, Bing, DuckDuckGo	NumPydoc, Pandasdoc, Private documentations

Table 2: Comparison between search tools for API selection.

suggestion. A comparison of these two types of search tools is given in Table 2.

Online Search Engine Tool Online search engine tools provide rich information about API usages. Human programmers share their experience in solving various programming problems on various community and tutorial websites such as *StackOverflow* (platform, 2023) and *datagy.io* (DataGY, 2023). They organize and summarize the API suggestions used for different problems. When other people encounter similar problems, search engines can use this information well. Commercial search engine tools such as *Google* (Google, 2023), *DuckDuckGo* (DuckDuckGo, 2023) can regard these community websites as knowledge resources and can provide helpful API suggestions, especially for those public libraries that are well-known and widely discussed.

Documentation Search Tool Since lesser-known public libraries or private libraries have few discussions on the online community websites, human programmers also turn to library documentations for API suggestions. Documentations are usually available for both public libraries and private libraries if one has access to the library. Therefore, it can provide rich information for any API usage scenario. Usually, API information in the documentation is usually provided in pairs of API and corresponding comments. We can use *BM25* (Robertson and Zaragoza, 2009) or other semantic similarity scores as search metrics to search for comments that meet the requirements and find the corresponding API as the final suggestion for coding.

These various search tools are helpful for selecting APIs. Inspired by the API search process of human developers, we aim to incorporate these two types of search tools into code generation models.

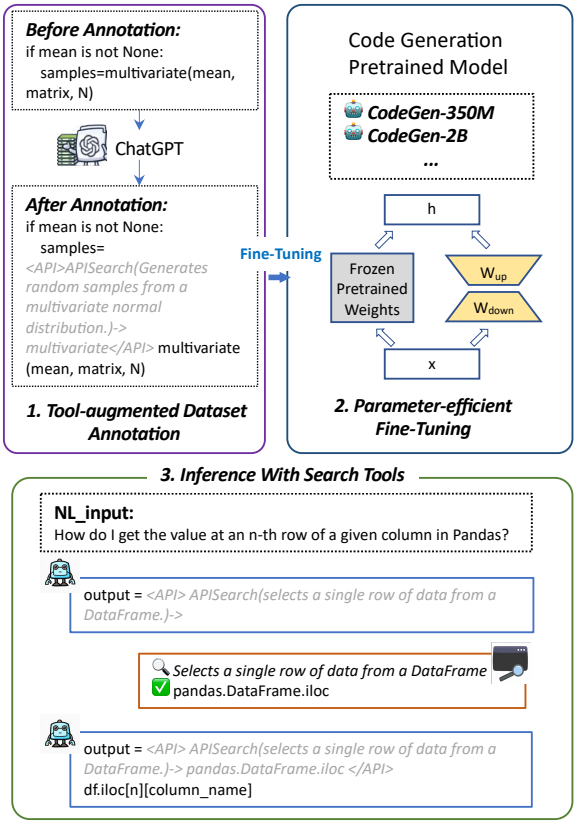

Figure 2: The pipeline of our approach ToolCoder. The pipeline has three main parts: ❶ Automatically Annotate Tool-augmented Dataset with ChatGPT, ❷ Parameter-efficient Fine-tune existing pre-trained code generation model with the annotated dataset, and ❸ Inference of the fine-tuned model enhanced with API search tools.

3 ToolCoder

The pipeline of ToolCoder is presented in Figure 2. There are three stages to implement ToolCoder: ❶ data annotation, ❷ fine-tuning, and ❸ inference. To deliver the readers an overall understanding, we first elaborate the tools we employed in ToolCoder (Section 3.1); then we illustrate the three modules in our pipeline in details, including the automatic annotation process for training data (Section 3.2, the fine-tuning process for ToolCoder (Section 3.3) and the inference algorithm to invoke search tools (Section 3.4).

3.1 API Search Tool Invocation

We first introduce the tools we adopted in ToolCoder. We develop two major categories of API search tools as introduced in Section 2.2. ❶ For those commonly used public libraries such as *numpy* and *pandas*, we employ ***DuckDuckgo*** as the search engine, which provides a cheaper and more convenient method compared to other search

engines like *Google*, *Bing*. We use the engine to search the relative content from several online community websites and extract the APIs with string regex matching mentioned in the context. Since these contents discuss the API in depth, more accurate API suggestions can be provided by the search engine. ❷ For those lesser-known or private library APIs, there is no relevant online information. We employ ***BM25*** score as our retrieval metric to search within the corresponding API documentation. We encapsulate these search interfaces so that ToolCoder can call search tools with high performance in a unified form. In our experiment, we control the search delay within *0.6s* to ensure high efficiency during the code generation process. We make an unified abstraction to unify the above-mentioned tools as below:

$$APISearch(query) \rightarrow answer \quad (1)$$

where $APISearch$ refers to the abstracted name of different API search tools, $query$ denotes the search query (functionality or requirement description), and $answer$ is the search result returned by the tools (*i.e.,*, an API recommendation).

Following our design, ToolCoder invokes a tool by generating the $APISearch(query) \rightarrow$ part, and wait for the tool's response, *i.e.,*, $answer$. In addition, to distinguish invocation of tools from normal source code, we surround the tool invocation with special tokens, starting with $\langle API \rangle$ and ending with $\langle /API \rangle$. All $\langle API \rangle$, $\langle /API \rangle$, and $\rightarrow$ are appended into the special token set in the code generation model's vocabulary. Please refer to the part in Figure 2 for a demonstrative example. We will elaborate the detailed algorithm in Section 3.4 later.

3.2 Automatic Data Annotation

To enable the model to use the API search tool, we fine-tune existing models with a dataset that includes the source code and associated tool call processes. As mentioned in Section 3.1, we abstract the search call process with the notation $\langle API \rangle\, APISearch(query) \rightarrow answer\, \langle /API \rangle$. However, such a dataset is not readily available. To address this issue, we propose to automatically augment an existing source code dataset with the tool call notation annotated by ChatGPT (gpt-3.5-turbo) (Chen et al., 2023; Ouyang et al., 2022). ChatGPT has already demonstrated excellent few-shot and even zero-shot learning ability in many different language learning tasks. This low-cost and efficient annotation method reduces the manual effort required to create private annotated datasets. A demonstrative example is presented in the left part of Figure 2.

Statistic		
Dataset Size		53,000
Avg. Annotation API		3.2
Avg. length (in words) before annotation		186.24
Avg. length (in words) after annotation		211.49
Proportion of some third-party libraries	*NumPy*	24%
	Pandas	13%
	TorchData	**0%**
	Private Libraries: *Monkey, BeatNum*	**0%**

Table 3: Statistics of the annotation dataset.

```
Your task is to add calls to a API Search Tool to a piece of
source code. You can use an API Search Tool to lookup
important third-party APIs from the document. The API Search
Tool should help you get information required to complete
the source code and select API. Use the format:
"<API>APISearch(query)->answer</API>". In the format,
"query" is the search input that describes the specific role
of the API required in this code, and "answer" is the search
output API. Here are some examples of API calls:

Input: B = np.reshape(A, (-1, 2))
Output: B = <API>APISearch(Gives a new shape to an array
without changing its data.)->np.reshape</API>np.reshape(A,
(-1, 2))

        (...another two Input-Output pairs...)

Input: {code}
Output:
```

Figure 3: An example of prompt used to generate API-augmented datasets for the API search tool.

We use the popular code corpus *CodeSearchNet-Python* (Husain et al., 2019) and ask the ChatGPT to annotate the tool-augmented dataset with the prompt in Figure 3. The details of the dataset construction are described in Appendix A. The final dataset, which will be used for subsequent fine-tuning, contains 53k of well annotated samples. Table 3 shows the statistics of the final annotated dataset. We list the proportion of some third-party library APIs in the dataset for reference in subsequent evaluation experiments.

3.3 Parameter-efficient Fine-tuning

In order to facilitate tool invocations in ToolCoder, fine-tuning is almost necessary. Even though we have a dataset annotated as describe in the previous subsection now, it is still relatively difficult to fine-tune a rather large model (*e.g.,*, CodeGen-2B with two billion parameters), due to limited computational power and high training efficiency requirements. To address such challenges, we propose to restrict the number of meta-trainable parameters and layers in the pre-trained model and adopting a parameter-efficient fine-tuning approach that can efficiently adapt pre-trained models to new task

Algorithm 1 Inference with API search tools

```
1:  procedure InferWithTool(model, input_nl, maxlen)
2:      Pass input_nl to the model and get predicted token
3:      output ← [token]
4:      i ← 0
5:      while i < maxlen do
6:          token ← the last token of output
7:          if token = ⟨API⟩ then
8:              query ← the following generated tokens between APISearch( and )→
9:              response ← Call API search tool with query
10:             Append ⟨API⟩APISearch(query)→response⟨/API⟩ to output
11:             i ← i+ length of the call process
12:         else
13:             Pass token to the model and get predicted token
14:             Append predicted token to output
15:             i ← i + 1
16:         end if
17:     end while
18:     return output
19: end procedure
```

types.

In our training setting, we we apply LoRA (Hu et al., 2022) to reduce trainable parameters. As a result, we only need to train 0.18% parameters in *CodeGen-350M* and 0.09% for *CodeGen-2B*. It makes it possible to efficiently fine-tune models on a consumer-level GPU, such as Nvidia GeForce RTX 2080 (11GB RAM). The parameter-efficient tuning strategy significantly reduces the training computational burden in our experiments. It can achieve results comparable to full-parameter training with less computational resources and time. We will give a detailed analysis of the ablation experiment in Section 5.3.

3.4 Inference enhanced with Tools

During inference, ToolCoder interacts with the tools in a similar manner of system calls in operating system. The code generation model initiates an invocation by generating the left half part of Eq. 1, and waits for the tool's response (*i.e.,*, the right half part of Eq. 1). We give a detailed pseudo-code description of the decoding process with API search tool procedure in Algorithm 1.

Specifically, during the inference stage, there are two modes of the code generation model – regular generation and tool invocation. The regular generation mode of ToolCoder is no difference than next token prediction (please refer to Line 12 in Algorithm 1). When an API needs to be generated, ToolCoder switches to the other tool invocation mode, by generating the $\langle API \rangle$ token (Line 7 in Algorithm 1). The model continue to decode the $APISearch$ token, the $\rightarrow$ token, along with the query between them. At this point, we interrupt the decoding process and call the API search tool to get a response. The search result is appended after the $\rightarrow$ token. Finally, the $\langle /API \rangle$ token is added to the output; ToolCoder completes one tool invocation and switches back to the regular mode. As the consequence, the code generation model is now capable to generate correct API calls according to the search result.

By leveraging API search tools in this way, ToolCoder can effectively address the challenge of selecting the right APIs and reduce the effort required by developers to find suitable APIs.

	Private Library		Public Library		
	MonkeyEval	BeatNumEval	TorchDataEval	PandasEval	NumpyEval
Known by Model	✖	✖	✖	✔	✔
Data Size	101	101	50	101	101
Avg. APIs	1.35	1.21	1.48	1.35	1.21
Avg. Tests	6.50	3.54	1.14	6.50	3.54

Table 4: Statistics of evaluated benchmarks.

4 Experimental Setup

4.1 Benchmark Datasets

Our experiments are conducted on two private library benchmarks and three public library benchmarks in Table 4. We also show whether our model known the corresponding library during training as *"Known by Model"*.

Private library benchmarks We use private library benchmarks to show how our approach can improve the accuracy of utilizing APIs that are unknown to the code generation models and lie beyond the scope of its knowledge. **MonkeyEval** (Zan et al., 2022a) is crafted by modifying all Pandas-related keywords to ensure that no information about the API names is leaked. **BeatNumEval** (Zan et al., 2022a) is crafted from the NumPy library. The pre-trained model has not seen the API in MonkeyEval and BeatNumEval, and the online search resources cannot provide any API-related information. So the API selection on these benchmarks will only rely on the API search tool we built on the documentation of these private libraries.

Public library benchmarks We use public library benchmarks to demonstrate how our approach can also enhance the accuracy of utilizing unfamiliar APIs within these open-source public

libraries. **TorchDataEval** (Zan et al., 2022a) is based on the TorchData library in Python. TorchData is a newly released library, which is unseen to the existing pre-trained code generation models. Therefore, this benchmark can also be used to demonstrate the generalization ability of our method on those APIs that are public but never seen by the code generation model. **PandasEval** (Zan et al., 2022b) is a domain-specific code generation benchmark for the Pandas library in Python. **NumpyEval** (Zan et al., 2022b) specifically targets the Numpy library in Python. Following the previous work, we use the metric pass rate ***pass@k*** (Chen et al., 2021) and set $k = \{1, 10\}$.

4.2 Baselines

We select six series of recent code generation models as baselines, including one of the most powerful models, GPT-3.5. These models can be divided into two categories: ❶ general models, such as **CodeT5** (Wang et al., 2021), **PyCodeGPT** (Zan et al., 2022b), **CodeGen** (Nijkamp et al., 2022), **GPT-3.5** (Chen et al., 2023; Ouyang et al., 2022) and ❷ API-oriented models, such as **CERT** (Zan et al., 2022b) and **CodeGenAPI** (Zan et al., 2022a). Details are shown in Appendix C.

4.3 Tools

When implementing the API search tool, we adopt in-site online search in *datagy.io* as well as *NumPy* (NumPy, 2023), *Pandas* (Pandas, 2023) and *TorchData websites* (TorchData, 2023) using the *DuckDuckGo*[1] for public library benchmarks. For private library benchmarks, we use provided *Monkey* and *BeatNum* library documentations to design an API search tool based on the BM25 algorithm. The tool's response for inference is considered as the first retrieved API. Other training and inference details are shown in Appendix D.

5 Results and Analyses

5.1 Private Library & API

We first evaluate our model on private library code generation (MonkeyEval and BeatNumEval). Results are shown in Table 5. *ToolCoder-DocTool* represents the performance of our model with the documentation search tool to generate code as these private libraries do not have relevant online resources.

[1]We choose *DuckDuckGo* because it provides a cheaper and more convenient API than other search engines such as *Google* and *Bing*.

Model	Param.	MonkeyEval		BeatNumEval		Avg.	
		pass@1	pass@10	pass@1	pass@10	pass@1	pass@10
General Models							
CodeT5	220M	0	0	0	0	0.000	0.000
CodeGen350M	350M	0.95	4.90	5.15	11.96	3.050	8.430
CodeGen2B	2B	1.59	5.94	5.94	11.88	3.765	8.910
API-oriented							
CodeGenAPI	350M	1.19	4.68	4.44	8.24	2.815	6.460
CodeGenAPI-retrieval	475M	**3.41**	**8.33**	5.90	11.79	4.655	10.060
CodeGen-retrieval	475M	2.46	6.35	6.65	13.68	4.555	10.015
GPT3.5	-	*2.47*	*8.91*	*6.68*	*17.82*	*4.575*	*13.365*
Ours							
ToolCoder-DocTool	350M	2.98	5.94	6.73	12.87	4.855	9.405
	2B	3.02	7.92	**6.93**	**13.86**	**4.975**	**10.890**

Table 5: Pass rate of models on private library benchmarks

The private library benchmarks are extremely hard for code generation models, such as CodeT5, which proves that the generation of API has particular challenges for code generation models. Our ToolCoder achieves the best average performance on these two private benchmarks. Compared with the base pre-trained model CodeGen-350M and CodeGen-2B, our model also greatly improves. It shows that documentation search tools can help code generation models select proper APIs during inference, thus improving the quality of the generated code. Results show that our proposed ToolCoder can assist the API generation process and enhance the ability of the code generation model.

When compared with the state-of-the-art API-oriented baselines, our model shows comparable performance. Note that *CodeGenAPI-retrieval* (Zan et al., 2022a) requires a well trained retriever model on the documentation. And their reported results are very unstable with hyperparameters. For instance, when changing the number of retrieval results, the pass@1 of *CodeGenAPI-retrieval* over MonkeyEval can vary from 1.94% to 3.41%. By contrast, our ToolCoder shows stable and comparable performance. Even compared with the most powerful model GPT3.5, our ToolCoder can achieve better results in some inference settings.

5.2 Public Library & API

We then evaluate our model on public library code generation (*TorchDataEval*, *NumpyEval* and *PandasEval*) and results are shown in Table 6. *ToolCoder-OnlineTool* represents the performance of our model with the online search engine tool to generate code.

It is worth noting that *TorchData* is a newly released library, which is unseen to the existing pre-trained code generation models except GPT3.5 and other API-oriented baselines. Our method can

Model	Param.	TorchDataEval		PandasEval		NumpyEval	
		pass@1	pass@10	pass@1	pass@10	pass@1	pass@10
General Models							
CodeT5	220M	0	0.1	0	0	0	0
PyCodeGPT	110M	3.80	14.00	12.75	37.62	18.04	38.61
CodeGen350M	350M	4.60	14.00	16.73	29.70	18.51	43.56
CodeGen2B	2B	7.00	18.00	30.69	42.57	29.10	53.46
API-oriented							
CERT-numpy	220M	2.20	14.00	16.03	27.72	31.47	46.42
CERT-pandas	220M	2.80	6.00	28.42	**48.04**	18.81	33.66
CodeGenAPI	350M	7.19	16.93	13.58	34.95	16.55	29.48
CodeGenAPI-retrieval	475M	10.41	23.50	11.25	28.61	12.67	27.32
CodeGen-retrieval	475M	7.52	16.36	9.54	29.02	18.30	35.12
GPT3.5	-	*6.00*	*24.00*	*30.09*	*33.16*	*58.41*	*66.21*
Ours							
ToolCoder-OnlineTool	350M	7.40	20.00	22.77	37.62	35.64	50.50
	2B	**11.80**	**24.00**	**31.68**	47.52	**41.58**	**55.44**

Table 6: Pass rate of models on public library benchmarks.

Dataset Setting	TorchDataEval		PandasEval		NumpyEval	
	pass@1	pass@10	pass@1	pass@10	pass@1	pass@10
ToolCoder-350M	7.40	20.00	22.77	37.62	35.64	50.50
original dataset	6.00	14.00	19.92	38.61	19.40	39.60
annotation w/o query	3.80	6.00	11.68	33.66	14.05	43.56

Table 7: Ablation studies on dataset settings against ToolCoder-350M

Training Setting	Training Time	Trainable Param.	TorchDataEval		PandasEval		NumpyEval	
			pass@1	pass@10	pass@1	pass@10	pass@1	pass@10
ToolCoder-350M	6h	0.65M	7.40	20.00	22.77	37.62	35.64	50.50
full-training	29h	350M	6.00	22.00	22.67	40.59	35.35	58.41

Table 8: Ablation studies on training settings against ToolCoder-350M.

achieve the best performance over all baselines on this benchmark. It indicates that the online search engine tool can well handle the unknown public API and assist models to generate accurate code.

Results also show that ToolCoder achieves the best average results on all three benchmarks. Our model achieves 1.39%, 3.26%, and 10.11% pass@1 improvement over the best API-oriented baseline on three benchmarks. Even when we control our model parameters to be smaller than the baselines as ToolCoder-350M, our model can still achieve excellent overall performances. Existing API-oriented models such as *CERT-numpy* and *CERT-pandas* mainly focus on training and inference on a library API code dataset, resulting in the failure of the same model to achieve good results on multiple API benchmarks. Our model shows stronger generalization ability and can be applied to various API libraries.

Combining the performance of our method on private library benchmarks, the average *pass@1* on five benchmarks of our two series of ToolCoder is 15.10%, 19.00%. For this average pass@1 metric, our ToolCoder outperforms all baselines by at least 6.21%. As for the average *pass@10*, our model outperforms by at least 9.64%. It is confident that our ToolCoder shows the overall best performance on various API generation scenarios.

5.3 Ablation

We further investigate the impact of different stage settings in our pipeline, including changing the dataset, training, and inference settings.

5.3.1 Dataset Setting

In Table 7, we replace our training dataset with the original dataset, which only contains the regular source code and without annotation, referring as *original dataset*. We also add an experiment to remove the content of the query in the search call so that its form becomes *APISearch()→answer*. During inference, we use the question description to search the API directly. We refer to this ablation as *annotation w/o query*. We also add the original *CodeGen-350M* model for comparison, which is not trained on the new dataset.

Results show that it is essential to generate the search query. When we discard the search query in the data construction and use the problem description for API search tools, we observe a drastic drop in the final results as *annotation w/o query*. We attribute it to the fact that the problem description is still far from the use of the specific API, so it is still difficult to select the appropriate API using the existing API search tools. We can also confirm that only fine-tuning on the original source code dataset can not help the model learn to select APIs. We compare the *CodeGen-350M* with the model trained on the *original dataset*. Results show that additional training on the code dataset does not significantly improve the model's performance. The key to our improvement is to annotate the API tool into the code dataset to teach the model to use external API search tools.

5.3.2 Training Setting

We performed ablation experiments with ToolCoder-350M on the training setting in Table 8. Our experiments compare the performance of two approaches: full parameter training, referred to as *full-training*. Our proposed method utilizes LoRA for parameter-efficient training. We evaluate their performance on public library benchmarks and recorded their training costs, including training time and parameters, using 2*2080 GPUs.

Results show that our fine-tuning strategy has

(a) On Public library benchmarks

Inference Setting	TorchDataEval		PandasEval		NumpyEval	
	pass@1	pass@10	pass@1	pass@10	pass@1	pass@10
OnlineTool-350M	7.40	20.00	22.77	37.62	35.64	50.50
NoTool-350M	6.00	16.00	20.19	35.64	33.76	46.53
OnlineTool-2B	11.80	24.00	31.68	47.52	41.58	55.44
NoTool-2B	7.50	20.00	31.38	44.55	38.71	54.45

(b) On Private library benchmarks

Inference Setting	MonkeyEval		BeatNumEval	
	pass@1	pass@10	pass@1	pass@10
DocTool-350M	2.98	5.94	6.73	12.87
NoTool-350M	0.29	0.99	1.68	4.95
DocTool-2B	3.02	7.92	6.93	13.86
NoTool-2B	0.79	2.97	2.77	8.91

Table 9: Ablation studies on inference settings.

almost no performance penalty compared with the regular *full-training*. On the public library benchmarks, the difference between the two pass@1 results is within 0.4%. The gap in these results is acceptable, considering the huge savings in training costs. In our experiment settings, our parameter-efficient fine-tuning strategy can reduce the training time from 29h to 6h and the training parameters from more than 350M to 0.65M. We only need to train 0.18% parameters in CodeGen-350M and 0.09% for CodeGen-2B. It makes it possible to efficiently fine-tune models on a consumer-level GPU, such as Nvidia GeForce RTX 2080 (11GB RAM).

5.3.3 Inference Setting

We perform ablation experiments on the inference setting in Table 9. We add experiments to disable the tool in our model. *NoTool* represents that we disable the tool for inference and use our trained model to directly generate an API based on the search query and complete the code. We compare with our original inference setting on public and private library benchmarks.

Experiments show that our external tools are essential in improving performance. On public library benchmarks, the online search engine tool improves pass@1 by 1.88%, 2.57%, 0.4% for ToolCoder-350M, and 2.87%, 0.29%, 4.3% for ToolCoder-2B. When considering private library benchmarks, the improvement is more significant. We find the model itself works poorly on private libraries. However, with the assistance of the documentation search tool, our model can choose a suitable private library API.

Another interesting observation is that the *NoTool* also achieves relatively good performance on public library benchmarks. We believe it comes from our dataset annotation process. The additional tool call process in the dataset can be seen as a way to think about and choose the API. The chain of thought in the annotation dataset can assist the code generation model in better understanding the functions of different APIs. However, for private libraries, since the knowledge of private libraries is not seen by the code generation model, this form of dataset annotation is challenging to bring improvements to the model. With proper API search tools enhanced, our ToolCoder can select API more accurately and improve further.

6 Related Work

Recently, some work has focused on selecting APIs during code generation. As discussed in Section 2.1, existing code generation models still struggle with selecting appropriate APIs for a given context, especially for private or lesser-known APIs. Existing work (Zan et al., 2022b,a; Zhou et al., 2023) has proposed some API-oriented code generation methods. They typically use a two-stage pipeline, where the first stage involves searching or generating related APIs and then using them to generate code. We pursue this research line and propose to leverage models and API search tools to automate API selection in coding practices. Tools have been demonstrated to assist large models in addressing numerous complex tasks (Schick et al., 2023; Nakano et al., 2021; Qin et al., 2023; Yao et al., 2022; Wang et al., 2023). Our approach has two advantages: ❶ Our method shows strong generalization ability. By setting an appropriate API search tool, our method can quickly adapt to any API-related code generation scenario. ❷ Our method does not require multi-stage generation. Instead, we integrate the API search tool into the decoding process, making our approach more flexible and allowing the API selection process to be closer to the specific code fragment being generated.

7 Conclusion

We propose ToolCoder, a novel approach to incorporate API search tools into the code generation process to assist models in generating unknown APIs. Experiments on public and private library code generation benchmarks show that our ToolCoder outperforms state-of-the-art methods. Our paper demonstrate the potential of incorporating programming tools into the code generation process, shedding light on this line of future work.

Limitation

Although we're just starting to explore this field, our work has some limitations that we're planning to fix soon:

First, constrained by our computational resources, we used smaller CodeGen series models as our base model. As our method is model-agnostic, we plan to use it to train stronger models in the future.

Next, we have simplified the API search tool to only return the names of the functions found. We have attempted to return the parameters of functions as part of the search results, but due to the flexibility of the Python language (some function parameters can be optional) and the complexity of the content in the search source, we did not yield desirable results in initial experiments. We will continue to explore how to design high-quality search tools to assist large models in generating those unknown APIs.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. volume abs/2107.03374.

Zekai Chen, Mariann Micsinai Balan, and Kevin Brown. 2023. Language models are few-shot learners for prognostic prediction. volume abs/2302.12692.

Google Colab. 2023. https://colab.research.google.com/.

DataGY. 2023. https://datagy.io/.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

DuckDuckGo. 2023. https://duckduckgo.com/.

Google. 2023. https://www.google.com/.

GPT-3.5. 2022. https://platform.openai.com/docs/models/gpt-3-5.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. volume abs/1909.09436.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. volume 55, pages 248:1–248:38.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2021. Webgpt: Browser-assisted question-answering with human feedback. volume abs/2112.09332.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. volume abs/2203.13474.

NumPy. 2023. https://numpy.org/doc/.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. volume abs/2203.02155.

Pandas. 2023. https://pandas.pydata.org/docs/.

StackOverflow platform. 2023. https://stackoverflow.com/.

Yujia Qin, Zihan Cai, Dian Jin, Lan Yan, Shihao Liang, Kunlun Zhu, Yankai Lin, Xu Han, Ning Ding, Huadong Wang, Ruobing Xie, Fanchao Qi, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2023. Webcpm:

Interactive web search for chinese long-form question answering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 8968–8988. Association for Computational Linguistics.

Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: Bm25 and beyond. volume 3, pages 333–389.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. volume abs/2302.04761.

TorchData. 2023. https://pytorch.org/data/.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 2609–2634. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. volume abs/2210.03629.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022a. When language model meets private library. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 277–288. Association for Computational Linguistics.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022b. CERT: continual pre-training on sketches for library-oriented code generation. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2369–2375. ijcai.org.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

A Details of Data Annotation

Our data annotation process can be divided into three parts: ❶ base dataset selection, ❷ prompt selection, and ❸ filter & clean. A demonstrative example is presented in the left part of Figure 2.

Base Dataset Selection For the base dataset, we choose to use the popular code corpus *CodeSearchNet-Python* (Husain et al., 2019) as the base dataset. It is a real-world programming dataset obtained from GitHub without additional annotations. This dataset is already used by many pre-trained code generation models (Wang et al., 2021; Nijkamp et al., 2022), so we can assure that our subsequent training will not affect the model's generalization performance on language generation and modeling ability as much as possible. We use a simple length filtering method and randomly choose nearly 60k function-level source code from this dataset as the base dataset for further annotation.

Prompt Selection Similar to ToolFormer (Schick et al., 2023), to help generate the annotated dataset, we need to provide a specific instruction for ChatGPT to specify its system role as a data annotator, as shown in Figure 3. To facilitate the quality of the generated datasets, we manually write three human-written input-output pairs as the demonstration part of the prompt with three libraries (*numpy*, *pandas*, and *matplotlib*). We choose these three libraries because ❶ they are widely utilized in Python programming and are familiar to most Python programmers, and ❷ they are frequently used in our base dataset. Based on our selected prompt and base dataset, we ask the ChatGPT to annotate the tool-augmented dataset. Specifically, as demonstrated in Figure 3, the role instruction ("Your task is ..."), the demonstration input-output pairs (hand-written), and the API (to be annotated) are concatenated, forming the prompt. ChatGPT then annotates the API of interest by following the instruction and imitateing the demonstrations. We generate one annotated data for each base sample. The overall automatic annotation process lasts for four days.

Filter and Clean After getting all the generated results from ChatGPT, we performed some filtrations on the results to remove those abnormal data samples. The nested API search calls are removed. We limit the number of API search calls in a sample to less than 5, and ensure that at least one of them is from a publicly available library. We also conduct rule-based correctness checks to ensure that the API search call is closely related to the specific code implementation. The final dataset, which will be used for subsequent fine-tuning, contains 53k of well annotated samples.

B Details of Parameter-efficient Fine-tuning

In our experiments, we apply LoRA (Hu et al., 2022) to reduce trainable parameters. Low-Rank Adaptation (LoRA) is a low-dimensional representation-based parameter-efficient tuning method. It injects trainable low-rank matrices into transformer layers to approximate the weight updates. For a pre-trained weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA represents its update with a low-rank decomposition $W + \delta W = W + W_{down}W_{up}$, where $W_{down} \in \mathbb{R}^{d\times r}, W_{up} \in \mathbb{R}^{r\times k}$ are tunable parameters. LoRA generally applies this update to the attention linear projection matrices in the multi-head attention sub-layers in Transformer. For a specific input x to the linear projection in multi-head attention, LoRA modifies the projection output h as:

$$h \leftarrow h + s \cdot xW_{down}W_{up}, \tag{2}$$

where $s \geq 1$ is a tunable scalar hyperparameter. The illustration of LoRA is shown in the middle part of Figure 2.

C Details of Baseline Models

We select six series of recent code generation models as baselines, including one of the most powerful models, GPT-3.5. These models can be divided into two categories: general models and API-oriented models.

General Models **CodeT5** (Wang et al., 2021) is an encoder-decoder pre-trained model for code-related tasks. It uses the identifier-aware pre-training task and has achieved SOTA results on many general code generation benchmarks. We use CodeT5-base with 220M parameters in our experiments. **PyCodeGPT** (Zan et al., 2022b) is a decoder-only pre-trained code generation model with 110M parameters. It is initialized with the GPT-Neo and is continually pre-trained with a large-scale code corpus in Python. **CodeGen** (Nijkamp et al., 2022) is a series of decoder-only pre-trained code generation models with parameters

varying from 350M to 16B. It casts code generation as a multi-turn conversation between a user and a system. CodeGen has shown strong ability on a variety of complex code generation tasks. Due to computational limitations, we use 350M and 2B versions in our experiments. **GPT-3.5** (Chen et al., 2023; Ouyang et al., 2022) is one of the most powerful generation models from OpenAI. We use the "*gpt-3.5-turbo*" model as it is the most cost-effective and performant model in the GPT3.5 family. As OpenAI states, it can be complemented with flexible natural language and programming language capabilities (GPT-3.5, 2022).

API-oriented models **CERT** (Zan et al., 2022b) is a generation approach designed for API-related code. CERT contains two modules: the sketcher and generator, each of which is fine-tuned independently with PyCodeGPT. It first predicts a sketch based on the NL description and generates the complete code based on the sketch. For each library, CERT requires a specially trained weight for generation. We use the released weight as two independent models: *CERT-numpy*, *CERT-pandas*. **CodeGenAPI** (Zan et al., 2022a) is another API-oriented code generation model. It uses a two-stage pipeline to generate code: given an NL description, CodeGenAPI firstly uses a retriever model initialized with BERT (Devlin et al., 2019) to find APIs from documents. Then it uses a generator initialized with CodeGen-350M to generate the complete code based on the retrieved API and problem description. We use the three released settings in their paper: *CodeGenAPI*, *CodeGen-retrieval*, and *CodeGenAPI-retrieval*. The first setting only uses the trained generator without retrieval, and the latter two use the best-performing top2 retrieval results to assist generation.

D Implementation Details

Training Our model is implemented in the Pytorch framework, and we perform all the experiments on four RTX 2080-11GB GPUs. We initialize our ToolCoder by leveraging pre-trained weights of CodeGen-350M and CodeGen-2B. The training batch size is set to 8. The learning rate is set to 1e-4. The total training epoch is set to 10. We use validation loss to determine the best checkpoint as the final model.

Inference During the model generation process, we use temperature sampling with $T = 0.8$ and limit the sample budget to 10. Each experiment is run three times with random seeds and then averaged for the final results.

E Discussion

Compared with ChatGPT Our ToolCoder achieves similar performance compared with ChatGPT, even outperforms this powerful model on those unknown APIs. The results can be unexpected to some extent, and we would like to discuss the reason for a bit. The whole process of generating an API for code generation models may be decomposed into three steps –❶ the generation model decides to use an API to achieve specific requirements, ❷ it collects knowledge (internal or external) of this API, and ❸ it generate an API call based on the knowledge. The knowledge gathering step is possibly the most important part during this process as it dictates the extent and reliability of the API knowledge that the model can tap into. For most code generation models like ChatGPT, this knowledge is based on its internal general understanding acquired during pre-training. However, these models falls short when applied to unknown API environments. In contrast, the API knowledge of ToolCoder comes from massive external sources – ToolCoder employs API search tools to ensure more accurate and extensible API knowledge. From the ablation study in Table 9 we can show that the performance improvement of ToolCoder is mostly from the tool utilization. With the proper tool assistant, even a rather small model can achieve great performance.

Besides, ToolCoder requires low deployment cost. Our model can be trained and developed with just a consumer-level GPU. We also conduct a time efficiency experiment over *TorchDataEval*. We use Google Colab (Colab, 2023) with a Tesla T4 GPU for evaluation. Our ToolCoder takes only 4.3 seconds to generate output for each sample, while ChatGPT requires 11 seconds. As a result, our model achieves better performance with less time cost.

F Case Study

We perform a case study analysis in Figure 4, which represents code snippets generated on public and private library benchmarks. From the examples, we obtain the following findings: ❶ The generated search query provides more fine-grained technical planning for the solution. The *NumpyEval* case

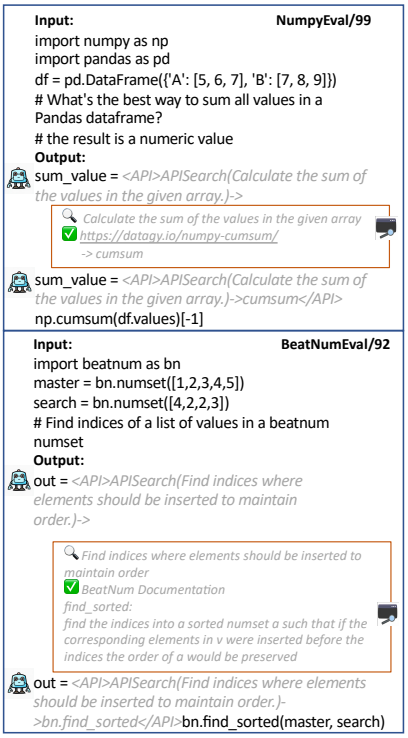

Figure 4: Cases produced by ToolCoder-2B, with online search engine tool on NumpyEval and documentation search tool on BeatNumEval respectively.

requires summing values in a dataframe, and the generated query breaks down the requirements, focusing first on summing arrays. It fills the gap between requirements and concrete APIs. ❷ The response of the search tools both play a crucial role in the generated code. The online search engine tool finds the proper API from the correct websites, and the documentation search tool finds the proper API by searching over the API comments. ❸ Our ToolCoder also can make necessary modifications based on the tool response. For example, the online search tool returns the response as *cumsum*, not directly defined in the input code. Our ToolCoder can add some components not in the response and generate the correct API *np.cumsum*.